INFORMATION AND COMPUTATION 118, 101–109 (1995)

Testing Language Containment for ω-Automata Using BDDs

HERVÉ J. TOUATI* AND ROBERT K. BRAYTON

Electrical Engineering and Computer Sciences Department, University of California, Berkeley, California 94720

AND

ROBERT KURSHAN

AT&T Bell Laboratories, Murray Hill, New Jersey 07974

ω-automata provide a well-established basis for the specification and verification of control-intensive systems. To verify that a system satisfies a given property ("task"), one specifies both the system and the task in terms of ω-automata, and then verifies that the ω-regular language of the system automaton is contained in that of the task automaton. This procedure, which is the basis of the COSPAN verification software, has been used in a number of commercial applications. However, its applicability is limited by the computational complexity of the ensuing language containment check, which tends to grow exponentially with the number of components in the system. While reduction techniques such as task decomposition and task-relative homomorphic reduction can greatly extend the complexity of systems which thus may be analyzed, there is a computational cost associated with such reductions as well. Moreover, the system complexity is the ultimate limiting factor. Recent advances in the manipulation of data-structures for binary decision diagrams (BDDs) have suggested that this data-structure may now facilitate checking language containment for far larger system models than has been hitherto possible. We have confirmed this by implementing new BDD-based language containment checks in COSPAN. We exhibit two such algorithms: one with a time advantage and the other with a space advantage. Each has increased significantly the size of system models which can be verified.

1. INTRODUCTION

ω-automata provide a well-established basis for the specification and verification of control-intensive systems such as communication protocols, cache coherency protocols, and switches. In such systems, the complex coordination among concurrently active components makes it difficult for a designer to foresee all possible behaviors. It is accordingly difficult to develop a simulator with test patterns which can catch all possible problems. While it has been customary to describe the behavior of such systems with linear-time temporal logic, the class of ω-automata is strictly more expressive than linear-time temporal logic, can simulate all regular languages, and is well adapted to the specification of fairness and liveness properties, as well as sequential properties such as counting (beyond the expressiveness of temporal logic). To verify that a system satisfies a given property, one specifies both the system and the property in terms of ω-automata, and then verifies that the ω-regular language of the system automaton is contained in that of an automaton which defines each property ("task") to be verified. In this sense, verification consists of a language containment check. This procedure, which is the basis of the COSPAN verification software, has been used in a number of commercial applications [12]. However, its applicability is limited by the computational complexity of the ensuing language containment check, which tends to grow exponentially with the number of system components. While reduction techniques such as task decomposition and task-relative homomorphic reduction [16–18] can greatly extend the complexity of systems which may thus be analyzed, there is a computational cost associated with such reductions as well. Moreover, the system complexity is the ultimate limiting factor. Recent advances [3] in the manipulation of data-structures for binary decision diagrams (BDDs) have suggested that these data-structures may now facilitate checking language containment for far larger system models than has hitherto been possible. We have confirmed this by implementing new BDD-based language containment checks in COSPAN [13]. We exhibit two such algorithms: one with a time advantage and the other with a space advantage. Each has increased significantly the size of system models which can be verified. For a selected problem with a scalable structure, we have shown that whereas with the non-BDD language containment algorithm, the verification running time grows exponentially with the size of the state space, and the effective size of a system which can be verified is 10^7 states, the BDD-based algorithms empirically demonstrate a running time which increases linearly with the size of the state space.

* Current address: DEC PRL, 85, avenue Victor Hugo, 92563 Rueil-Malmaison Cedex, France.

0890-5401/95 $6.00

This allows verification of a system model scaled to 10^{18} states, in roughly the same time as was required to verify the 10^7-state model using explicit enumeration (the non-BDD algorithm). Moreover, commercial system models with state spaces orders of magnitude larger have been verified using these new algorithms (these models are not scalable, and hence the performance of non-BDD and BDD-based algorithms cannot be directly compared). A more detailed report on the implementation of these algorithms is found in [13].

Although not without drawbacks (BDD-based algorithms run slower in some cases, and are extremely sensitive to implementation issues such as variable-ordering and memory de-allocation), the BDD-based algorithms described in this paper have proved to be an important tool for extending the power of language-containment checking.

The class of ω-regular languages is related to the class of regular languages in that they both are defined by finite state automata. Unlike regular languages, which are composed of sets of strings, ω-regular languages consist of sets of (infinite) sequences. Whereas automaton acceptance for strings is defined simply in terms of a set of "final" states, there are a variety of such acceptance conditions for sequences. Each such condition carries implications in terms of expressive power and computational complexity of associated decision procedures [6]. A pair of classes of ω-automata found to be especially convenient for modeling concurrent systems are L-automata and L-processes [16–18, 12], each of which define all ω-regular languages. For both L-automata and L-processes, the acceptance condition is defined by a set of *recur* edges, and a set of *cycle* sets of respective states. A sequence is accepted by an L-automaton if and only if it has a run of the automaton which either traverses a given recur edge infinitely often or is such that the set of states visited infinitely often is contained in some cycle set. For an L-process, the acceptance condition is the negation of this: a sequence is accepted if and only if it has a run which traverses no recur edge infinitely often, and is such that the set of states visited infinitely often is contained in no cycle set; thus, in the case of the L-process, the acceptance condition may be understood as an "exception" condition. L-processes provide a natural mechanism to model a "system" process, as the exception condition may be interpreted as a "fairness" property, excepting "unfair" sequences which, for example, never leave a set of states (a cycle set). L-automata provide a natural mechanism to model system properties (tasks) which are to be verified; for example, a "liveness" or "eventuality" property may be defined in terms of sequences which traverse a given set of recur edges infinitely often. If we want to check that a process is granted access to a resource infinitely often (liveness property) provided that the process does not remain in a set of states where it never requests the resource again (fairness property), we impose the fairness condition on an L-process S, which models the system, and express the liveness property in terms of an L-automaton T. Verification that the system satisfies the property thus is represented as verifying the language containment $\mathcal{L}(S) \subset \mathcal{L}(T)$. When T is deterministic, this can be checked in time linear in the number of edges of S and of T [16]. Moreover, every ω-regular language $\mathcal{L}$ is equal to some intersection $\mathcal{L} = \bigcap \mathcal{L}(T_i)$ for a finite number of deterministic L-automata $T_1, \ldots$ [17], so $\mathcal{L}(S) \subset \mathcal{L}$ can be verified by checking $\mathcal{L}(S) \subset \mathcal{L}(T_i)$ for each i. Therefore, from the point of view of expressiveness, it is enough to be able to verify $\mathcal{L}(S) \subset \mathcal{L}(T)$ for deterministic L-automata T.

The algorithm for verifying $\mathcal{L}(S) \subset \mathcal{L}(T)$ for an L-process S and a deterministic L-automaton T proceeds by constructing an L-process P from S and T whose language satisfies $\mathcal{L}(P) = \mathcal{L}(S) \backslash \mathcal{L}(T)$ (whose size is the product of the sizes of S and T), and testing the emptiness of $\mathcal{L}(P)$ [17]. Therefore, it is enough to be able to test for emptiness the language of an (arbitrary) L-process P. This may be done by enumerating the states explicitly, finding the set of strongly connected components of the transition graph of P with recur edges deleted, and checking that each strongly connected component is contained in some cycle set [17].

However, when S is large, so is P. In this paper, we present BDD-based algorithms for testing emptiness of the language of an L-process, thereby giving a BDD-based algorithm for testing language containment of ω-automata. The use of BDDs facilitates the manipulation of sets of states (the set of satisfiability of a Boolean function describing the state transition predicate). Since the domain of operation is sets of states rather than individual states, the feasibility of these algorithms is governed not by the size of the underlying state space, but by the complexity of the underlying Boolean functions. Empirical evidence indicates that for the class of problems which arise naturally as verification problems, the use of the BDD-based algorithms can significantly extend the set of feasible models. The success of these algorithms lies in recent techniques developed for the manipulation of binary decision diagrams [3, 2].

A special case of the language containment problem is to show that given two deterministic finite state machines, every input/output sequence possible in one is possible in the other. Such a test is performed by computing the product of the two machines, and demonstrating that for each product state reachable from a pair of component initial states, the component outputs are identical. This can be done very effectively with binary decision diagrams using the implicit enumeration technique introduced by Coudert *et al.* [8]. Unfortunately this method cannot be generalized directly to perform a language containment check for ω-regular languages, which is significantly more complex.

A formulation of a language containment problem in terms of μ-calculus was given in [5], although this technique

applies only to deterministic Büchi automata (which cannot express all ω-regular languages). The first algorithm we present here appeared in an earlier version of this paper [22]; the second algorithm, based upon a formula of Emerson and Lei [10], was inspired by [5].

Verification based upon ω-automata bears a close relationship to temporal logic model checking (cf. LTL [9], CTL [7]). The linear time logic LTL and extensions of the branching time logic CTL may be used to describe fairness and liveness properties of non-terminating finite state systems. However, LTL is strictly less expressive than ω-automata. CTL cannot express general fairness, while its extensions which can have model-checking complexity which is exponential in the size of the formula. (A hybrid form of model-checking which combines CTL and ω-automata has the same complexity as that for CTL.) Finally, the existential path quantification in CTL severely limits the extent of homomorphic reduction which is possible.

We begin, in Section 2, by reviewing properties of BDDs and techniques for implicit state enumeration using BDDs. Then in Section 3 we show how to represent the "system" process in terms of its components, as well as the recur edges and the cycle sets, using BDDs. In Section 4 we present a new verification algorithm for testing ω-automaton language containment, and in Section 5 we present an alternative algorithm derived from the formula of Emerson and Lei in the propositional μ-calculus [10, 5]. In Section 6 we discuss some important implementation issues, including a comparison of the two algorithms. Finally in Section 7 we discuss error-reporting: the problem of presenting a counter-example in case of a verification failure.

2. TERMINOLOGY AND NOTATION

2.1. *Binary Decision Diagrams*

A binary decision diagram (BDD) [19, 1, 3] is a data structure that represents a Boolean function $f: B^n \to B$, where $B = \{0, 1\}$. In the form introduced by Bryant [3], BDD representations have three main advantages: they are reasonably small for a large class of interesting Boolean functions, they are canonical for a given ordering of the input variables, and they can be directly manipulated to perform all basic Boolean operations efficiently. All other known representations of Boolean functions fail to have at least one of these properties. Truth tables, for example, are of exponential size. Boolean networks are efficient representations in terms of storage requirements and Boolean operations, but are poor representations for tautology checking, which is a vital primitive operation for verification algorithms.

A BDD representing a Boolean function f is a directed acyclic graph (DAG) whose leaves are the constant nodes 0 and 1, and whose internal nodes can only implement an if-then-else operation, where the conditional part is restricted to being the value of an input variable. In addition, on any path from the root to a leaf, a conditional or branching variable can appear at most once, and in some specified order [3]. For a given variable ordering, a BDD can be seen as the DAG obtained from the Shannon decomposition of f for that ordering by maximal sharing of common subexpressions.

2.2. *States, Sets of States, and State Relations*

In this paper, we suppose that a finite state system is specified as a network of interacting finite state machines (L-processes). The automaton defining a system property under test (task) also is part of that description, as already discussed in the Introduction. The resulting state space is a Cartesian product of respective state spaces of component L-processes. The Boolean representation of a state is the conjunction of the representations of the component states.

Finite state machines usually are represented in terms of symbolic states and symbolic inputs and outputs, not in terms of binary variables. To be able to represent a set of states S with BDDs, we encode symbolic variables with a small set of binary variables using an encoding that minimizes the number of BDD nodes [21], and we represent as a BDD the *characteristic function* of S. For simplicity, we associate a set with its characteristic function.

We also need to represent with BDDs the state transition relation $T(x, y)$ of the system (given in terms of *current* state x and *next* state y). Using a transition relation instead of a transition function has two main advantages: it allows us to handle non-determinism, and it allows us to abstract away from the representation the specific conditions under which a given transition is possible. The transition relation can be interpreted as a set of pairs of states; we represent its characteristic function as a BDD. Other binary state relations also can be represented that way. One binary relation of particular interest is the transitive closure $R(x, y)$ of the transition relation $T(x, y)$.

It is also useful in some cases to extend the BDD representation of the transition relation to incorporate input variables i. In that case the transition relation becomes a ternary relation $T(x, i, y)$, which can be interpreted as a set of triplets, whose characteristic function again is represented as a BDD.

2.3. *Primitive Operations*

Boolean Operations. We can operate directly on BDDs to perform all the usual set operations: union, intersection, complementation. These operations actually are performed on the characteristic functions of the corresponding sets; thus the Boolean `or` operator ($+$) denotes union, the Boolean `and` operator ($\cdot$) denotes intersection, and the

Boolean not operator (not $S = \bar{S}$) denotes complementation.

Projection. To use binary relations, we also need to be able to perform a projection onto a set of variables. For example, to compute the set R_1 of states reachable in one step from a given set of states R, where R and R_1 are represented by their characteristic functions $R(x)$ and $R_1(x)$, we need to be able to compute the set $\{y \mid \exists x\, T(x, y) \cdot R(x) = 1\}$. We overload the existential quantifier to denote this projection operator as follows: $R_1(y) = \exists x(T(x, y) \cdot R(x))$. The projection operator can be computed directly on a BDD representation [5] by decomposing it into simple operations as illustrated below:

$$\exists(x_{i_1}, ..., x_{i_k})\, f = \exists x_{i_1} \cdots \exists x_{i_k} f$$

$$\exists x_i f(x_1, ..., x_n) = f_{x_i}(x_1, ..., x_n) + f_{\bar{x}_i}(x_1, ..., x_n)$$

$$f_{x_i}(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$$

$$f_{\bar{x}_i}(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$$

Fixed Point Computations. The last operators we need to describe our algorithm are fixed point operators. Let F be a function mapping the power set 2^S of some finite set S into itself. For simplicity, we associate a set with its characteristic function. Throughout, we assume that F is *monotone*, i.e., $A \subseteq B$ implies that $F(A) \subseteq F(B)$. We define lfp(F), the least fixed point of F, and gfp(F), the greatest fixed point of F, as follows:

$$\text{lfp}(F) = \bigcap_{A = F(A)} A$$

$$\text{gfp}(F) = \bigcup_{A = F(A)} A.$$

Let F^0 be the identity function, and let $F^{i+1} = F \circ F^i$ be the function obtained by applying F to the result of F^i. Since F is monotone and operates on a finite set, one can easily show that there is some integer $i \geq 0$ such that $F^i(\varnothing) = F^{i+1}(\varnothing) = \text{lfp}(F)$, and similarly some integer $j \geq 0$ such that $F^j(S) = F^{j+1}(S) = \text{gfp}(F)$. The check for termination of fixed point computations is a tautology check, for which the use of BDDs is an important asset.

A simple example of a fixed point computation is the computation of the set of states $R(x)$ reachable from a given set of states $I(x)$. R is equal to lfp(F), where F is the following function operating on sets c of states:

$$F(c)(x) = I(x) + \exists y(T(y, x) \cdot c(y)). \tag{1}$$

A related fixed point computation is the computation of the set of states $I(x)$ than can reach a given set of states $R(x)$. $I(x)$ is equal to lfp(F), where F is the function

$$F(c)(x) = R(x) + \exists y(T(x, y) \cdot c(y)). \tag{2}$$

A more complex example is the computation of the transitive closure $R(x, y)$ of the transition relation $T(x, y)$. In that case, we compute the fixed point of a function operating on relations (sets of pairs of states). $R(x, y)$ is equal to lfp(F), where F is the following function:

$$F(c)(x, y) = T(x, y) + \exists z(T(x, z) \cdot c(z, y)). \tag{3}$$

3. FROM AUTOMATA TO BDDS

We assume that the language containment problem $\mathcal{L}(S) \subset \mathcal{L}(T)$ has been translated into a language emptiness problem for an L-process P, as described in the Introduction. Since the L-process S which defines the system model commonly is defined as a product of component processes, this product will be manifest in the representation of P as well. In this section we describe how to construct the transition relation for the L-process P, defined in terms of component processes.

The system modelled by the L-process S is assumed to be "closed," that is, to include a model of the environment. Such a system model does not have any external inputs or external outputs. Started in some set of initial states, at each global state (state of P), respective component outputs (which may be several, chosen non-deterministically) are used as inputs to the respective system components at that state. Each component L-process may depend only upon a small subset of the collective outputs. Since these signals are either inputs or outputs depending upon the context, to avoid confusion we refer to them as *selections*. The component L-processes which define the system model S may be interpreted as Moore-like machines with non-deterministic outputs, and acceptance conditions represented by recur edges and cycle sets, as described in the Introduction.

3.1. *Building the Transition Relation*

Let $P_1, ..., P_n$ be the finite state L-processes composing the system model S. The state space of the system is the Cartesian product of the state spaces of the individual processes. The transition relation of the system is the Boolean and of the transition relations of the individual processes.

For a given process P, we build first the partial transition relation $PT_P(x_P, i, y_P)$. $PT_P(x_P, i, y_P) = 1$ if and only if state y_P of P is reachable in one step from state x_P under selection i. PT_P not only depends on the present state and next state of process P, but also on the selection i of the entire system. The selection variables are not externally generated: they are the aggregate outputs of all processes at state $x = (x_{P_1}, ..., x_{P_n})$. Thus we also need to represent, for each process P, the possible selections of that process. This is done by building another relation, $O_P(x_P, i_P)$, where i_P represents the selection variables that correspond to the

outputs of process P. $O_P(x_P, i_P) = 1$ if and only if the selection i_P can be generated by process P at state x_P. The full transition relation of process P is the conjunction of these two relations:

$$T_P(x, i, y) = PT_P(x_P, i, y_P) \cdot O_P(x_P, i_P).$$

The global transition relation $T(x, y)$ for the entire system then is computed by taking the conjunction of the transition relations of its component processes, and applying the existential quantification operator to T to remove any dependency on the selection variables i:

$$T(x, y) = \exists i \left(\prod_{j=1}^{n} T_{P_j}(x_{P_j}, i, y_{P_j}) \right). \tag{4}$$

3.2. *Representation of Recur Edges*

We represent recur edges [17] implicitly in the transition relation. We add a dummy Boolean selection variable r to the transition relation. This variable is not removed by existential quantification during the computation of the transition relation. Thus the transition relation is actually a function of three sets of variables: $\hat{T}(x, r, y)$. $\hat{T}(x, 1, y) = 1$ if and only if y is reachable from x in one step across a recur edge, and $\hat{T}(x, 0, y) = 1$ if and only if y is reachable from x in one step across an edge that is not a recur edge.

3.3. *Representation of Cycle Sets*

Cycle sets [17] are sets of states and are directly represented as BDDs. We use the notation $(\mathscr{C}_1(x), ..., \mathscr{C}_n(x))$ to designate the Boolean functions which define the cycle sets.

4. ALGORITHM AND JUSTIFICATION

As we have reduced the language containment check to an emptiness check for L-processes, this is logically equivalent to checking that every reachable cycle in the state transition graph of the system either contains a recur edge or is entirely contained in one of the cycle sets [16]. We present here an algorithm that can perform this check through implicit enumeration. Neither states nor cycles nor recur edges are enumerated explicitly. The algorithm proceeds in five steps:

1. We first compute the set of reachable states $R(x)$ from a set of initial states $I(x)$.

2. We remove the recur edges from the transition relation and restrict the transition relation to reachable states, by using $T(x, y) = R(x) \cdot \hat{T}(x, 0, y)$.

3. We compute the transitive closure $R(x, y)$ of $T(x, y)$. A pair (x, y) satisfies $R(x, y) = 1$ if y can be reached from x in one or more transitions.

4. For each cycle set $\mathscr{C}_i$, we use $R(x, y)$ to enumerate implicitly all states in some cycle not contained in $\mathscr{C}_i$. The union of these sets of states is called $NC_{\mathscr{C}_i}$ (for *Not Contained in* $\mathscr{C}_i$).

5. We compute the intersection of the sets $NC_{\mathscr{C}_i}$. The language containment check succeeds if and only if this intersection is empty.

These steps are elaborated below.

4.1. *Computing Reachable States*

Computing the set of reachable states is done by computing the least fixed point of the function F of equation (1). In this computation, we use all edges, whether they are recur edges or not.

4.2. *Removing Recur Edges*

Once the set of reachable states $R(x)$ has been computed, we simplify the transition relation by removing all states that are not reachable. We also remove all the recur edges from the transition relation. This can be done *only after* the set of reachable states has been computed; otherwise we may miss some cycles that do not contain any recur edges but are only reachable from the initial states through recur edges. We simplify the transition relation as follows:

$$T(x, y) = R(x) \cdot \hat{T}(x, 0, y). \tag{5}$$

Since we have removed from the transition relation all the states that are not reachable and all the state transitions that occur only through recur edges, the cycles remaining in the state graph represented by $T(x, y)$ are exactly the cycles that are reachable from an initial state and do not contain any recur edge. At this point, to complete the language containment check, we only need to verify that every cycle in $T(x, y)$ is contained in one of the cycle sets. This technique of *implicitly* eliminating cycles containing recur edges was introduced in [16].

4.3. *Computing the Transitive Closure of the Transition Relation*

We have discussed earlier how to compute the transitive closure $R(x, y)$ of the transition relation $T(x, y)$ as the least fixed point of the function F given in Eq. (3).

We can deduce cyclic relationships from $R(x, y)$. A pair of states (x, y) belongs to $R(x, y)$ if and only if y can be reached from x in one or more transitions. Consequently, a pair of states (x, y) belongs to $R(x, y) \cdot R(y, x)$ if and only if x is reachable from y and y is reachable from x. This is equivalent to saying that there is a cycle containing both x and y, or that x and y belong to the same strongly connected component of the state transition graph.

4.4. *Implicit Enumeration of Unaccepted Cycles*

For each cycle set $\mathscr{C}_i$ we compute the set $NC_{\mathscr{C}_i}$,

$$NC_{\mathscr{C}_i}(x) = \exists y(R(x, y) \cdot R(y, x) \cdot \overline{\mathscr{C}_i}(y)), \tag{6}$$

where $\overline{\mathscr{C}_i}$ is the complement of the cycle set $\mathscr{C}_i$. A state x belongs to $NC_{\mathscr{C}_i}$ if and only if there exists a cycle containing x not entirely contained in $\mathscr{C}_i$. We now show that the set $NC_{\mathscr{C}_i}$ implicitly enumerates all the unaccepted cycles with respect to $\mathscr{C}_i$, i.e., all the cycles not contained in $\mathscr{C}_i$.

THEOREM 4.1. *Every reachable cycle is contained in at least one of the cycle sets $\mathscr{C}_i$ iff the set NC is empty, where*

$$NC = \bigcap_{1 \leqslant i \leqslant n} NC_{\mathscr{C}_i}. \tag{7}$$

The proof of this result relies on the following lemma:

LEMMA 4.2. *Let SCC be the set of strongly connected components of the state transition graph. Then $NC_{\mathscr{C}_i}$ is the union of all the strongly connected components in SCC not entirely contained in $\mathscr{C}_i$:*

$$NC_{\mathscr{C}_i} = \bigcup_{\substack{c \in SCC \\ c \not\subseteq \mathscr{C}_i}} c. \tag{8}$$

Proof of Lemma 4.2. Let $c \in SCC$ be such that $c \not\subseteq \mathscr{C}_i$. Let x be any element of c, and let y be an element of $c - \mathscr{C}_i$. Then the pair (x, y) belongs to the set $R(x, y) \cdot R(y, x) \cdot \overline{\mathscr{C}_i}(y)$ and thus x belongs to $NC_{\mathscr{C}_i}$. Conversely, let x be an element of $NC_{\mathscr{C}_i}$. There is an element y such that x and y belong to the same cycle c and y does not belong to $\mathscr{C}_i$. This cycle c is contained in the strongly connected component C containing x. C is notcontained in $\mathscr{C}_i$ and thus x belongs to $\bigcup_{c \in SCC,\, c \not\subseteq \mathscr{C}_i} c$. ∎

Proof of Theorem 4.1. We first prove that if $NC \neq \varnothing$ then there is a cycle not contained in any of the cycle sets $\mathscr{C}_i$. Let x be an element of NC; x is contained in a (unique) strongly connected component C, which is a cycle. From the lemma we deduce that C is not contained in any of the $\mathscr{C}_i$. Conversely, suppose that there is a cycle c not contained in any of the cycle sets $\mathscr{C}_i$. Let C be the strongly connected component containing c. *A fortiori* C is contained in none of the $\mathscr{C}_i$. Thus, according to the lemma, C is contained in all of the sets $NC_{\mathscr{C}_i}$, whose intersection is therefore not empty. ∎

5. THE EMERSON–LEI FORMULA

In [10] Emerson and Lei introduced a restricted form of the propositional mu-calculus [15] that is at least as expressive as many of the commonly used temporal logics, including FCTL (CTL extended with fairness constraints [7, 5]). The formula Emerson and Lei introduced to express an FCTL fairness constraint in the mu-calculus can be adapted to handle cycle sets in the context of an ω-regular language containment check [20], in replacement of the second part of the previous algorithm (Sections 4.3 and 4.4).

The Emerson–Lei formula computes a set NC_μ that we will prove is equal to the set of reachable states from which the set NC of Eq. (7) can be reached. NC_μ is thus empty if and only if NC is, and can be computed in place of NC to detect unaccepted cycles. NC_μ is obtained as the greatest fixed point of the following function F:

$$F(c)(x) = c(x) \cdot \prod_{i=1}^{n} \exists y(T(x, y) \cdot \mathrm{lfp}(G_i^c)(y)). \tag{9}$$

Evaluation of F requires the computation of n least fixed points, one per function G_i^c, for each iteration of F. For a given set c, G_i^c is defined as follows:

$$G_i^c(d)(x) = c(x) \cdot (\overline{\mathscr{C}_i}(x) + \exists y(T(x, y) \cdot d(y))). \tag{10}$$

A state x belongs to $\mathrm{lfp}(G_i^c)$ if there is a path starting from x and entirely contained in c whose end point does not belong to $\mathscr{C}_i$ (cf. Eq. (2)).

THEOREM 5.1. *NC_μ is the set of reachable states from which NC can be reached.*

We will need the following lemmas in the proof of Theorem 5.1:

LEMMA 5.2. $\mathrm{lfp}(G_i^c) \subseteq c$.

LEMMA 5.3. *Let c be a fixed point of F, i.e., $F(c) = c$. Then each point of c has a successor in c.*

Proof of Lemma 5.3. Let x be an element of c. Since $F(c)(x) = c(x) = 1$, in particular there is a state y such that $T(x, y) = 1$ and $\mathrm{lfp}(G_1^c)(y) = 1$. Lemma 5.2 and the second equality imply that y is an element of c, and the first equality implies that y is a successor of x. ∎

LEMMA 5.4. *If c is a fixed point of F, then from each point of c and each integer i, $1 \leqslant i \leqslant n$, there is a path entirely included in c that ends at a point in $\overline{C_i}$.*

Proof of Lemma 5.4. Let x be an element of c, and i an integer between 1 and n. Since $F(c)(x) = c(x) = 1$, in particular there is a state y such that $T(x, y) = 1$ and $\mathrm{lfp}(G_i^c)(y) = 1$. Since y belongs to $\mathrm{lfp}(G_i^c)$, there is a path from y to a point outside $\mathscr{C}_i$ that is entirely included in c, which extends to a path from x to a point outside $\mathscr{C}_i$ entirely included in c. ∎

Proof of Theorem 5.1. Let NC' be the set of all reachable states from which NC can be reached. We want to prove that $NC_\mu = NC'$.

To show that $NC' \subseteq NC_\mu$, since $NC_\mu = \text{gfp}(F)$, it is sufficient to prove that $F(NC') = NC'$. Since $F(c) \subseteq c$ is always true, we simply have to prove that $F(NC') \supseteq NC'$. For a given i, $\text{lfp}(G_i^{NC'})$ is the set of states from which there is a path entirely included in NC' whose end point does not belong to $\mathscr{C}_i$. Since any point in NC' reaches NC, and any point in NC is on a cycle not entirely included in $\mathscr{C}_i$, we have $NC' \subseteq \text{lfp}(G_i^{NC'})$, and with Lemma 5.2 we obtain $NC' = \text{lfp}(G_i^{NC'})$. Thus

$$F(NC') = NC'(x) \cdot \exists y(T(x, y) \cdot NC'(y)).$$

In other words, $F(NC')$ contains all the points of NC' that have successors in NC'. Since every point in NC' has a successor in NC', we have $F(NC') \supseteq NC'$.

Conversely, let x be a state in NC_μ. We are going to show that x can reach a cycle not included in any of the $\mathscr{C}_i$, which is enough to show that x belongs to NC'. Let G be the directed graph obtained by restricting the state transition graph to NC_μ. Because NC_μ is a fixed point of F, by Lemma 5.3 each state in NC_μ has a successor in NC_μ. Thus each node in G has a successor. Let DG be the DAG obtained from G by collapsing all the connected components of G. Let C be a leaf of DG that is reachable from x. Let x_0 be an arbitrary state in C. Because NC_μ is a fixed point of F, by Lemma 5.4 there is a state y_1 reachable in one step from x_0 such that from y_1, there is a path entirely contained in NC_μ that ends outside $\mathscr{C}_1$. This path has to be entirely contained in C, because C corresponds to a leaf of DG. Let x_1 be the end point of that path. Similarly, we can find a path from x_1 to x_2, entirely contained in C, such that the end point x_2 does not belong to $\mathscr{C}_2$. This path can be continued to a path entirely contained in C that contains, for each $\mathscr{C}_i$, a state not included into $\mathscr{C}_i$. Thus C is an unaccepted cycle, which proves that $C \subseteq NC$. Since C is reachable from x, we have proved that x belongs to NC'. ∎

6. IMPLEMENTATION ISSUES

6.1. *Computation of the Transition Relation*

The storage requirement for the computation of the transition relation often can be reduced dramatically by exploiting the following algebraic identity: $\exists(x, y)(f(x, y) \cdot g(y)) = \exists y(g(y) \cdot \exists x f(x, y))$ [23]. This reduction is obtained by ordering the computation of the partial products in Eq. (4) in order to facilitate the elimination of variables by existential quantification.

6.2. *Computation of the Reachability Relation*

The most computationally expensive step in the algorithm of Section 4 is the computation of the transitive closure of $T(x, y)$. It relies on the fixed point computation of Eq. (3), which requires three sets of simultaneously active state variables, instead of two for Eqs. (1), (9), and (10). To make efficient use of BDDs in this context, we need to find effective variable ordering heuristics to use in Eq. (3).

Variable Ordering. A simple heuristic is to reflect in the variable ordering the initial decomposition of the system as a network of finite state machines. For each finite state machine, the binary variables encoding the present state and the next state of the machine are interleaved using an ordering heuristic such as the one introduced in [23] or the more recent techniques of [14]. The global variable ordering is deduced from the variable ordering of each machine by concatenation of the orderings. For example, for a system composed of n machines $(M_i)_{1 \leq i \leq n}$, each having only two possible states, and for a given ordering σ of these machines, if the present state of M_i is represented by one binary variable x_i and the next state of M_i is represented by one binary variable y_i, the global variable ordering we use is given by $(x_{\sigma(1)}, y_{\sigma(1)}, x_{\sigma(2)}, y_{\sigma(2)}, \ldots, x_{\sigma(n)}, y_{\sigma(n)})$. To choose the order of concatenation σ, we use a heuristic that clusters the component finite state machines in order to minimize the interaction between clusters. Reducing interaction between clusters should lead to smaller information transfer between levels in the BDDs and thus to smaller BDDs [4]. This clustering can be achieved in a variety of ways, e.g., using the heuristic of [23] and is better done before selection variables are removed from the transition relation. To introduce a third set z of variables, as needed in Eq. (3), we simply interleave the z variables with the x and y variables. If only one state variable is needed per machine, and if we suppose that $\sigma(i) = i$ for simplicity, the ordering would be $(x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n)$.

Iterative Squaring. The least fixed point of the predicate transformer F of Eq. (3) can be computed with logarithmically fewer iterations using *iterative squaring* [5]. This method consists of computing $\text{lfp}(F)$, where F is the following function:

$$F(c)(x, y) = T(x, y) + \exists z(c(x, z) \cdot c(z, y)). \qquad (11)$$

6.3. *Implicit Enumeration of Unaccepted Cycles*

The second most difficult computation in the algorithm of Section 4 is Eq. (6). The BDD size of $R(x, y)$, which represents the transitive closure of the transition relation $T(x, y)$, is expected to be significantly larger than the BDD size of $C_i(x)$, which represents only a set of states. As a heuristic, it is better to compute the product $R(y, x) \cdot \overline{\mathscr{C}_i}(y)$ first, and to compute the second Boolean `and` combined with the projection operator in a single pass over the BDDs, reducing the need for temporary storage [5]. Using this technique, we do not need to store the product $R(x, y) \cdot R(y, x)$ at any given point of the computation.

A second optimization is based on the use of the *generalized cofactor*, as defined in [23] (see below). This operator was initially proposed by Coudert *et al.* in [8] and called the *constraint* operator. The generalized cofactor of a Boolean function f by a Boolean function c is a Boolean function f_c that has, in most cases, a smaller BDD representation than f and can be substituted for f in some contexts.

Given a Boolean function $f=(f_1, ..., f_m)$: $B^n \to B^m$ and a subset of B^n represented by its characteristic function c, the generalized cofactor $f_c=((f_1)_c, ..., (f_m)_c)$ is a function from B^n to B^m whose range is equal to the image of c by f. For a single output function f: $B^n \to B$, the pair (f, c) can be interpreted as an incompletely specified function whose onset is $f \cdot c$ and whose don't care sets $\bar{c}$. Under this interpretation, the generalized cofactor f_c can be seen as a heuristic to select a representative of the incompletely specified function (f, c) that has a small BDD representation. The generalized cofactor f_c depends in general on the variable ordering used in the BDD representation. If c is a cube (i.e., product term), the generalized cofactor f_c is equal to the usual cofactor of a Boolean function, and is, in that case, independent of the variable ordering.

DEFINITION 1. Let c: $B^n \to B$ be a non-null Boolean function. We define the mapping π_c: $B^n \to B^n$ as follows:

$$\text{if} \quad c(x)=1 \quad \pi_c(x)=x$$
$$\text{if} \quad c(x)=0 \quad \pi_c(x)=y$$

where $c(y)=1$ and $d(x, y)$ is minimal, for $d(x, y)=\sum_{1 \le i \le n} |x_i - y_i| \, 2^{n-i}$.

LEMMA 6.1. *π_c is the projection that maps a minterm x to the minterm y in the onset of c which is closest to x according to the distance function d. The particular form of d guarantees the uniqueness of y in this definition, for any given ordering of x.*

DEFINITION 2. Let f: $B^n \to B$ and c: $B^n \to B$, with $c \neq 0$. The generalized cofactor of f with respect to c, denoted by f_c, is the function $f_c = f \circ \pi_c$ (i.e., $f_c(x)=f(\pi_c(x))$). If f: $B^n \to B^m$, then f_c: $B^n \to B^m$ is the function whose components are the cofactors by c of the components of f.

The generalized cofactor can be computed very efficiently in a single bottom-up traversal of the BDD representations of f and c [23].

In particular, the generalized cofactor satisfies the following two properties:

$$(f \cdot g)_c = f_c \cdot g_c$$
$$\exists y(f(x, y) \cdot c(y)) = \exists y(f_c(x, y)).$$

We can use the generalized cofactor to help reduce the cost of the computation of Eq. (6). As before, the Boolean and is computed in combination with the projection

$$NC_{\mathscr{C}_i}(x) = \exists y(R_{\overline{\mathscr{C}}_i(y)}(x, y) \cdot R_{\overline{\mathscr{C}}_i(y)}(y, x)).$$

(Here, $R(x, y)$ and $R(y, x)$ are viewed as functions of y, for fixed x.)

6.4. *Comparison with the Emerson–Lei Formula*

The Emerson–Lei formula has one major advantage over the algorithm of Section 4: it does not require the computation of the transitive closure $R(x, y)$ of the transition relation $T(x, y)$. Actually, it does not even require the explicit building of the transition relation $T(x, y)$ since we can use the technique introduced in [23] to compute fixed points without the need of an explicit representation for $T(x, y)$. On the other hand, it requires the computation of n least fixed points at each iteration of a greatest fixed point computation.

In contrast, the algorithm of Section 4 trades off memory for speed. It requires the memory-intensive computation of the reachability relation $R(x, y)$, but once this relation is built, no additional fixed point computation is required. It is faster than the method based upon the Emerson–Lei formula, when $R(x, y)$ can be built. Both methods have been implemented [13], providing empirical evidence of their respective merits.

7. FAILURE REPORTING

In a design verification environment, e.g., COSPAN [12], we need to report a failure along with an error track from an initial state to an unaccepted cycle. To make the BDD approach practical, we also have to develop a means of reporting such an error track in case of failure.

One possibility is to use BDD manipulations directly to report an error track. This can be done as follows. During the computation of reachable states, as in Section 4.1, we store the sets R_k that contain exactly the states reachable from an initial state in k transitions or less. In case of failure, we compute the smallest k for which the set $R_k \cap (\bigcap_{1 \le i \le n} NC_{\mathscr{C}_i})$ is not empty. We pick an arbitrary state x_k in this set, and compute the strongly connected component containing x_k, which is $c(x)=R(x, x_k) \cdot R(x_k, x)$. We can report $c(x)$ as an unaccepted cycle; $c(x)$ is guaranteed to be a cycle with the shortest distance to an initial state. To report a failure track leading to $c(x)$, we compute iteratively a sequence of states $(x_{k-1}, ..., x_0)$ such that x_{i+1} is reachable from x_i in one transition and x_i is in R_i, by picking for x_i any state x contained in $R_i(x) \cdot T(x, x_{i+1})$. In particular, x_0 is guaranteed to be an initial state. The sequence $(x_0, ..., x_k)$ can then be used as an error track leading to the unaccepted cycle $c(x)$. We obtain with

this algorithm an unaccepted cycle closest to the set of initial states and a shortest error track for this unaccepted cycle, but we do not necessarily find the smallest unaccepted cycle.

A natural question is whether we could report the smallest unaccepted cycle containing x_k. The answer is: probably not if the number of cycle sets grows too large. The problem of finding the smallest cycle containing x_k that contains a point in each of the sets $\overline{\mathscr{C}_i}$ can be reformulated as the following decision problem: given a strongly connected directed graph, n sets of nodes $(A_1, ..., A_n)$, and an integer B, is there a tour that visits each of the sets A_i and is of length B or less? This decision problem can easily be shown to be NP-complete by reducing to it a version of the traveling salesman problem [11]. In practice we can resort to a bounded depth branch and bound algorithm to find a short unaccepted cycle containing x_k. The branching part of the search selects which set $A_i = \overline{\mathscr{C}_i}$ to visit next.

8. CONCLUSION

We have presented a set of techniques for checking ω-regular language containment using BDDs. The techniques apply to a system described as a network of interacting ω-automata. Our algorithms can support the verification of liveness and fairness properties, and are compatible with formal abstraction mechanisms. The most difficult part of the verification procedure is the handling of fairness constraints (cycle sets). We have presented a new algorithm to handle fairness constraints. We compared this algorithm to one based upon a formula due to Emerson and Lei [10] which can be adapted to the context of ω-regular language containment [20]. The former algorithm is simpler than the one based upon Emerson and Lei's formula, and is faster but less memory efficient.

9. ACKNOWLEDGMENTS

The authors gratefully acknowledge many helpful discussions with T. Kam, B. Lin, R. Rudell, A. Sangiovanni-Vincentelli, and H. Savoj. Special thanks are due to K. McMillan for pointing out the mu-calculus algorithm of Section 5 and to T. Kam for running the suites of tests used to compare the BDD and non-BDD algorithms. This project is supported in part by the Defense Advanced Research Projects Agency under Contract N00039-87-C-0182 and National Science Foundation under Contract MIP-8719546.

Received November 20, 1991; final manuscript received September 14, 1993

10. REFERENCES

1. Akers, S. B. (1978), Binary decision diagrams, *IEEE Trans. Comput.* **C-27**, 506–516.
2. Brace, K. L., Bryant, R. E., and Rudell, R. L. (1990), Efficient implementation of a BDD package, *in* "27th ACM/IEEE Design Automation Conference."
3. Bryant, R. E. (1986), Graph based algorithms for Boolean function manipulation, *IEEE Trans. Comput.* **C-35**, 677–691.
4. Bryant, R. E. (1991), On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication, *IEEE Trans. Comput.* **40**, 205–213.
5. Burch, J. R., Clarke, E. M., McMilan, K. L., Dill, D. L., ankd Hwang, L. J. (1990), Symbolic model checking: 10^{20} states and beyond, *in* "Logic in Computer Science," Springer-Verlag, Berlin/New York.
6. Choueka, Y. (1974), Theories of automata on ω-tapes: A simplified approach, *J. Comput. System Sci.* **8**, 117–141.
7. Clarke, E. M., Emerson, E. A., and Sistla, P. (1986), Automatic verification of finite-state concurrent systems using temproal logic specifications, *ACM Trans. Programming Language Systems* **8**, 244–263.
8. Coudert, O., Berthet, C., and Madre, J. C. (1989), Verification of sequential machines based on symbolic execution, *in* "Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems, Grenoble."
9. Emerson, E. A. (1990), Temporal and modal logic, *in* "Handbook of Theoretical Science B," Vol. 16, pp. 995–1072, Elsevier, Amsterdam.
10. Emerson, E. A., and Lei, C. L. (1986), Efficient model checking in fragments of the propositional mu-calculus, *in* "Symposium on Logic in Computer Science," pp. 267–278, IEEE Press, New York.
11. Garey, M. R., and Johnson, D. S. (1979), "Computers and Intractability: A Guide to the Theory of *NP*-Completeness," Mathematical Sciences Series, Freeman, New York.
12. Har'El, Z., and Kurshan, R. P. (1990), Software for analytical development of communication protocols, *AT&T Technical J.*, (January 1990), pp. 45–59.
13. Hojati, R., Touati, H., Kurshan, R. P., and Brayton, R. K. (1993), "Efficient ω-Regular Language Containment," Lecture Notes in Computer Science, Vol. 663, pp. 396–409, Springer-Verlag, Berlin/New York.
14. Jeong, S. W., Plessier, B., Hachtel, G. D., and Somenzi, F. (1991), Variable ordering for FSM traversal, *in* "International Workshop on Logic Synthesis."
15. Kozen, D. (1983), Results on the proportional mu-calculus, *Theoret. Comput. Sci.*, 333–354.
16. Kurshan, R. P. (1987), "Reducibility in Analysis of Coordination," Lecture Notes in Control and Information Sciences, Vol. 103, pp. 19–39, Springer-Verlag, Berlin/New York.
17. Kurshan, R. P. (1990), "Analysis of Discrete Event Coordination," Lecture Notes in Computer Science, Vol. 430, pp. 414–453, Springer-Verlag, Berlin/New York.
18. Kurshan, R. P. (1994), "Computer-Aided Verification of Coordinating Processes," Princeton Univ. Press, Princeton, NJ.
19. Lee, C. Y. (1959), Representation of switching circuits by binary-decision programs, *Bell Systems Technical J.* **38**, 985–999.
20. McMillan, K. (1991), Personal communication.
21. Srinivasan, A., Kam, T., Malik, S., and Brayton, R. K. (1990), Algorithms for discrete function manipulation, *in* "IEEE International Conference on Computer-Aided Design," pp. 92–95.
22. Touati, H. J., Brayton, R. K., and Kurshan, R. P. (1991), Testing language containment for ω-automata using BDDs, *in* "International Workshop on Formal Methods in VLSI Design, Miami," ACM SIGDA.
23. Touati, H. J., Savoj, H., Lin, B., Brayton, R. K., and Sangiovanni-Vincentelli, A. (1990), Implicit state enumeration of finite state machines using BDDs, *in* "ICCAD'90."